Attorney's Docket No.: [redacted]0559-268001 / P9023

APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: BREAKPOINT METHOD FOR PARALLEL HARDWARE THREADS IN MULTITHREADED PROCESSOR

APPLICANT: DEBRA BERNSTEIN, SERGE KOMFELD, DESMOND R. JOHNSON, DONALD HOOPER, JAMES D. GUILFORD AND RICHARD D. MURATORI

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL688219376US

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

12-21-00
Date of Deposit

Derek W. Norwood
Signature

Derek W. Norwood
Typed or Printed Name of Person Signing Certificate

Breakpoint for Parallel Hardware Threads in Multithreaded Processor

TECHNICAL FIELD

This invention relates to breakpoint method for parallel hardware threads in a multiprocessor.

BACKGROUND

Parallel processing is an efficient form of information processing of concurrent events in a computing process. Parallel processing demands concurrent execution of many programs in a computer, in contrast to sequential processing. That is, in general all or a plurality of the stations work simultaneously and independently on the same or common elements of a problem.

Parallel processing involves multiple processors. The execution path of a microprocessor within a parallel system is highly pipelined.

In a parallel processor where many threads of execution can run simultaneously, there is a need for debugging software running on selected threads. Debugging may be used to determine a cause (or causes) of errors in the processing threads, and to correct the errors.

Debug methods implement breakpoints in software by a combination of inserting traps and single stepping. When the target program contains multiple threads of execution, a debug

method that is not carefully implemented may miss breakpoints and be less than helpful to the developer.

DESCRIPTION OF DRAWINGS

FIG. 1 is a block diagram of a communication system employing a hardware based multithreaded processor.

FIG. 2 is a block diagram of a microengine functional unit employed in the hardware based multithreaded processor of FIGS. 1 and 2.

FIG. 3 is a block diagram of an interrupt register.

FIG. 4 is a block diagram of a hardware based multithreaded processor adapted to enable a breakpoint method.

FIG. 5 is a flow chart of a breakpoint method for selectively stopping parallel hardware threads from a debug console.

DETAILED DESCRIPTION

Referring to FIG. 1, a communication system 10 includes a parallel, hardware based multithreaded processor 12. The hardware-based multithreaded processor 12 is coupled to a bus 12, such as a PCI bus, a memory system 16 and a second bus 18. The system 10 is especially useful for tasks that can be broken into parallel subtasks or functions. Specifically, hardware-based multithreaded processor is useful for tasks that are bandwidth oriented rather than latency oriented. The hardware-based multithreaded processor 12 has multiple microengines 22, each with multiple hardware controlled

threads (also referred to as contexts) that can be simultaneously active and independently work on a task.

The hardware-based multithreaded processor 12 also includes a central processor 20 that assists in loading microcode control for other resources of the hardware-based multithreaded processor 12 and performs other general purpose computer type functions, such as handling protocols, exceptions, extra support for packet processing where the microengines pass the packets off for more detailed processing, such as in boundary conditions. In an embodiment, the processor 20 is a Strong ARM® (ARM is a trademark of ARM Limited, United Kingdom) based architecture. The processor 20 has an operating system. Through the operating system, the processor 20 can call functions to operate on microengines 22. The processor 20 can use any supported operating system, preferably a real-time operating system. For a processor 20 implemented as a Strong ARM® architecture, operating systems such as Microsoft NT Real-Time, VXWorks and μCUS, a freeware operating system available over the Internet, can be used.

As mentioned above, the hardware-based multithreaded processor 12 includes a plurality of functional microengines 22a-f. Functional microengines (microengines) 22a-f each maintain a number of program counters in hardware and states associated with the program counters. Effectively, a corresponding plurality of sets of threads can be simultaneously active on each of the microengines 22a-f while only one is actually operating at any one time.

In an embodiment, there are six microengines 22a-f, as shown. Each of the microengines 22a-f has capabilities for processing four hardware threads. The six microengines 22a-f operate with shared resources, including memory system 16 and bus interfaces 24 and 28. The memory system 16 includes a synchronous dynamic random access memory (SDRAM) controller 26a and a static random access memory (SRAM) controller 26b. SRAM memory 16a and SRAM controller 26a are typically used for processing large volumes of data, e.g., processing of network payloads from network packets. The SDRAM controller 26b and SDRAM memory 16b are used in a networking implementation for low latency fast access tasks, e.g., accessing lookup tables, memory from the core processor, and so forth.

The six microengines 22a-f access either the SDRAM 16a or SRAM 16b based on characteristics of the data. Thus, low latency, low bandwidth data is stored in and fetched from SRAM 16b, whereas higher bandwidth data for which latency is not as important, is stored in and fetched from SDRAM 16b. The microengines 22a-f can execute memory reference instructions to either the SDRAM controller 26a or SRAM controller 26b.

Advantages of hardware multithreading can be explained by SRAM or SDRAM memory accesses. As an example, an SRAM access requested by a thread_0, from a microengine will cause the SRAM controller 26b to initiate an access to the SRAM memory 16a. The SRAM controller 26b controls arbitration for the SRAM bus, accesses the SRAM 16a, fetches the data from the SRAM 16a, and returns data to a requesting microengine 22a-f.

During an SRAM 26b access, if the microengine, e.g. microengine 22a, had only a single thread that could operate, that microengine would be dormant until data was returned from the SRAM 26b. By employing hardware context swapping within each of the microengines 22a-f, the hardware context swapping enables only contexts with unique program counters to execute in that same microengine. Thus, another thread, e.g., thread_1 can function while the first thread, e.g., thread_0, is awaiting the read data to return. During execution, thread_1 may access the SDRAM memory 26a. While thread_1 operates on the SDRAM unit, and thread_0 is operating on the SRAM unit, a new thread, e.g., thread_2 can now operate in the microengine 22a. Thread_2 can operate for a certain amount of time, until it needs to access memory or perform some other long latency operation, such as making an access to a bus interface. Therefore, simultaneously, the processor can have a bus operation, an SRAM operation and SDRAM operation all being completed or operated upon by one microengine 22a and have one or more threads available to process more work in the data path.

Each of the microengines 22a-f includes an arbiter that examines flags to determine the available threads to be operated upon. Any thread from any of the microengines 22a-f can access the SDRAM controller 26a, SRAM controller 26b or bus interface. The memory controllers 26a and 26b each include a number of queues to store outstanding memory reference requests. The queues either maintain order of

memory references or arrange memory references to optimize memory bandwidth. For example, if a thread_0 has no dependencies or relationship to a thread_1, there is no reason that thread_1 and thread_0 cannot complete their memory references to the SRAM unit 26b out of order. The microengines 22 a-f issue memory reference requests to the memory controllers 26a and 26b. The microengines 22a-f flood the memory subsystems 26a and 26b with enough memory reference operations such that the memory subsystems26a and 26b become the bottleneck for processor 12 operation. Microengines 22a-f can also use a register set to exchange data.

The processor 20 includes a RISC core 50, implemented in a five-stage pipeline performing a single cycle shift of one operant or two operants in a single cycle, provides multiplication support and 32-bit barrel shift support. This risc core 50 is a standard Strong Arm® architecture, but is implemented with a five-stage pipeline for performance reasons. The processor 20 also includes a 16-kilobyte instruction cache 52, an 8-kilobyte data cache 54 and a prefetch stream buffer 56. The core processor performs arithmetic operations in parallel with memory writes and instruction fetches. The processor 20 interfaces with other functional units via the ARM defined ASB bus. The ASB bus is a 32-bit bi-directional bus.

Referring to FIG. 2, an exemplary one of the microengines, microengine 22f, is shown. The microengine 22f includes a control store 70, which, in an implementation

includes a RAM of here 1,024 words of 32-bits each. The RAM stores eight microprogram. The microprogram is loadable by the processor 20. The microengine 22f also includes controller logic 72. The controller logic 72 includes in instruction decoder 73 and program counter units 72 a-d. The four program counters 72 a-d are maintained in hardware. The microengine 22f also includes context event switching logic 74. Context event switching logic 74 receives messages from each of the shared resources, e.g., SRAM 16a, SDRAM 16b, or processor 20, control and status registers, and so forth. These messages provide information on whether a requested function has completed. Based on whether or not a function requested by a thread (or context) has completed a signaled completion, the thread needs to wait for that completion signal, and if the thread is enabled to operate, then the thread is placed on an available thread list (not shown). The microengine 22f can have a maximum four threads available in the example of FIG 2.

In addition to event signals that are local to an executing thread, the microengines 22 employ signaling states that are global. With signaling states, an executing thread can broadcast a signal state to all microengines 22. Receive request available signal, any and all threads in the microengines can branch on these signaling states. These signaling states can be used to determine the availability of a resource or whether a resource is due for servicing.

The context event logic 74 has arbitration for the four threads in the example. In an embodiment, the arbitration is a round robin mechanism. Other techniques could be used, including priority queuing or weighted fair queuing. The microengine 22f also includes an execution box (EBOX) datapath 76 that includes an arithmetic logic unit 76a and general purpose register set 76b. The arithmetic logic unit 76a performs arithmetic and logical functions as well as shift functions. The register set 76b has a relatively large number of general purpose registers. General purpose registers are windowed so that they are relatively and absolutely addressable.

The microengine 22f also includes a write transfer register stack 78 and a read transfer stack 80. These registers 78 and 80 are also windowed so they are relatively and absolutely addressable. The write transfer register stack 78 is where write data to a resource is located. Similarly, the read register stack 80 is for returned data from a shared resource. Subsequent to, or concurrent with data arrival, an event signal from the respective shared resource, e.g., the SRAM controller 26b, the SDRAM controller 26a, or processor 20 will be provided to context event arbiter 74 which will then alert the thread is available or has been sent. Both transfer register banks 78 and 80 are connected to the execution box 76 through a datapath. In an implementation, the read transfer register 80 has sixty-four registers and the write transfer register 78 has sixty-four registers.

Each microengine 22a-f supports multi-threaded execution of four contexts. One reason for this is to allow one thread to start executing just after another thread issues a memory reference and must wait until that reference completes before doing more work. This behavior is critical to maintaining efficient hardware execution of the microengines, because memory latency is significant. Stated differently, if only a single thread execution was supported, the microengines would sit idle for a significant number of cycles waiting for references to return and thus reduce overall computational throughput. Multithreaded execution involves all microengines to hide memory latency by performing useful, independent work across several threads.

When errors occur in software running in one or more of the threads of execution, there is a need for debugging the software running on selected threads to determine a cause (or causes) of the errors and to aid a software developer to correct the errors.

Referring to FIG. 3, as described above, a multiprocessor 500 has many threads of execution, threads 502a-d for example, that are supported by the hardware. The core processor 504 runs the ARM instruction set, and executes one software thread at a time, in accordance with its corresponding program counter. In an embodiment, there are twenty-four hardware-supported threads. Any of these threads can interrupt the core processor 504 by issuing a write operation to a specified

control and status register (CSR) referred to as an interrupt register 506.

In a normal interrupt, a hardware-supported thread, thread 502a for example, executes a write to the interrupt register 506 that carries ten bits of immediate data. A ``1'' in bits 6:0 is shifted left by the value of the hardware thread identification and inserted into the interrupt register.

Referring to FIG. 4, the interrupt register 506 is a 32-bit register contained within the controlling processor 504. Bits 9:7 are designated breakpoint bits 600 and bits 6:0 are designated as a thread vectors 602. A non-zero value of the breakpoint bits 600 indicates the interrupt is a breakpoint interrupt. When a breakpoint occurs, a value of bits 9:7 of intermediate data is inserted into the breakpoint bits 600. A ``1'' in bits 6:0 of the intermediate data is shifted left by the value of the hardware thread identification to generate a thread vector; the thread vector is inserted into the thread vector bits 602 of the interrupt register 506.

At any time, more than one thread can issue a normal interrupt or an interrupt indicating a breakpoint. If the interrupt indicates a breakpoint, the core processor 504 traps to an interrupt handling routine 508 in an interrupt handler 510, where the interrupt register 506 is read. If the breakpoint field is non-zero, it can be used to narrow the source of the breakpoints to three groups of threads. Bits

23:0 are used to identify which threads have raised an interrupt.

The core processor 504 is linked to a remote user interface 511. A user (not shown) inputs a source code line to be breakpointed in one of the microengines using the remote user interface 511. The source code line is sent to the central processor 504. The central processor 504 searches a breakpoint database 512 in a debug library 514 to determine whether the instruction corresponding to the source code line can be breakpointed. Since there are certain cases where breakpointing is not allowed, i.e., a trap in the code cannot be inserted at this position in the code to signal a breakpoint. For example, if two instructions must be executed in succession, i.e., a register of one is used in the very next cycle in the next, the first instruction cannot be breakpointed. This is because the software breakpoint inserts a branch to displace the breakpointed instruction, thus separating the two instructions.

After searching its breakpoint database 512 and determining that an instruction may be breakpointed, the core processor 504 invokes a remote procedure call (RPC) referred to as SetBreakpoint to the debug library 514. The SetBreakpoint RPC identifies which microengine (which instruction code) to insert the breakpoint into, which program counter (PC) to breakpoint at, which microengine threads (also referred to as contexts) to enable breakpoint for, and which microengines to stop if the breakpoint occurs.

The function definition for the set breakpoint RPC to the debug library is as follows:

```
typedef struct   uDbg_Bkpt{
     void (*callback) (unsigned short bkptId,
unsigned char ueng, unsigned char ctx, void *userdata);
     short id; //out:the assigned ID of the breakpoint
     unsigned short ctx; //out: context at breakpoint
     unsigned shortμAddv; //in:micro-store address
     unsigned char μEng; //in:microengine at breakpoint
     void *usrData; //in:pointer to user data
     uDbg_CTX_T breakIfCtx; //in:break if equal to ctx
     unsigned int stpOnBrkUengMask; //in:microengines to
stop on break
} uBbg_Bkpt_T;
```

The debug library 514 generates a breakpoint routine by modifying a template of several instructions, inserts the breakpointing instruction and inserts a branch to the location of the instruction after the breakpoint instruction. The breakpoint instruction is essentially a branch instruction to the breakpoint routine.

An example breakpoint template follows:

```
unsigned int breakInst[NUM_BKPT_INST] = {
NOP, // may be replaced by: BR! = CTX, or BR=CTX
NOP, // may be replaced by: BR! = CTX, or BR=CTX
DISABLE_CTX, // disable context
NOP, // nop:
NOP,
NOP,
FAST_WR, // fast_wr[0x80, ireg] = breakpoint interrupt
CTX_ARB, //ctx_arb[voluntary]
0x9A0007B0, // statement at breakpoint placeholder--to be
altered
BR}; //br[pr] - to be altered to breakpoint +1
```

After the breakpoint instruction is inserted into the selected microengine, the microengine with this breakpoint code is resumed or restarted. If program execution within the microengine gets to the breakpoint program counter, program

execution branches to the breakpoint routine and interrupts the core processor 504 with a breakpoint interrupt. If a breakpoint is set, the interrupt handler 510 calls the debug library 514. The interrupt handler 514 can quickly handle breakpoints by reading the program counters of the interrupting threads. The debug library 514 stops the selected threads (and bus ports) for the breakpoint. The debug library 514 determines which thread(s) sent the breakpoint interrupt. The debug library 514 replies to the user on the remote user interface 511, allowing the user to examine the state of the saved threads (and ports). The user selects a resume command that initiates resume remote procedure call to the debug library 514. The resume identifies which microengines (and ports) to resume. Upon receiving this resume RPC, the debug library 514 insures the thread that the sent the breakpoint interrupt will be next to start by setting a control and status register with context enable. The debug library 514 restarts the microengines (and bus ports) indicated by the resume command.

Referring to FIG. 5, a breakpoint process 600 for selectively stopping parallel hardware threads from a remote user interface includes receiving 602 source code line to be breakpointed in a microengine. The process 600 fetches data 604 within its database to determine whether the source code line in the microengine may be breakpointed. If not, the process 600 signals 606 an error to the user. If the source code line can be breakpointed, the process 600 invokes 608 a

setbreakpoint RPG to the debug library. The setbreakpoint RPC identifies which microengine (or bus port) to insert the breakpoint into, which program counter to breakpoint at, which microengine threads to enable breakpoints for, and which microengines (or bus ports) to stop if a breakpoint occurs.

The process 600 causes the debug library to generate 610 a breakpoint routine by modifying a template of several instructions, to insert 612 the breakpoint instruction and to insert 614 a branch to the location after the breakpoint instruction.

After the breakpoint routine is inserted 612, the process 600 causes the microengine with the inserted breakpoint routine to resume 616. If program execution in the microengine gets to the breakpoint PC, program execution branches 618 to the breakpoint routine and interrupts 620 the core processor with the breakpoint interrupt.

The process 600 causes the interrupt handler to call 622 the debug library. The debug library stops 624 the selected threads (and bus ports) for the breakpoint and determines 626 which microengine sent the interrupt. The process 600 causes the debug library to display 626 information to the user.

The process 600 receives 630 a resume command from the user and in response sets the context enable bit for the selected microengine. Setting of the context enable bit starts the microengine back into normal program execution outside of the breakpoint routine.

An embodiment of the invention has been described. Nevertheless, it will be understood that various modifications may be made without departing from the spirit and scope of the invention. For example, reset, start, pause and resume remote procedure calls can be used with the controlled stop/start of selected microengines (and bus ports). The RPC ``Reset'' stops the selected microengines (and bus ports) immediately, whereas the RPC ``Start'' starts the microengines at program counter 0 (and starts the bus ports). The RPC ``Pause'' stops selected microengines (and bus ports) at a safe, i.e., non-destructive stopping point. The RPC ``Resume'' starts the microengines at their current program counters (and starts the bus ports). Accordingly, other embodiments are within the scope of the following claims.